**为什么要做资源管理？**

美术资源和游戏数据很多，不可能都放内存里面

**资源文件和内存对象的关系？**

uasset文件和 umap等在打包时会根据平台需要被cook成更小的平台专用文件，放在后缀是pak的压缩包中，游戏运行时，程序就会挂载解压缩这些pak包，然后加载包中的资源文件来使用

程序在用资源的时候，并不是直接在用这些文件。而是把这些文件转化成UObject或其他程序可以用的内存对象，如网格资源文件，程序用的实际是UStaticMesh对象、 而把资源文件，转变为内存里的UObject对象，就是资源管理做的事情。

**对于UE4来说，这个过程大概有这几个步骤：**

1，读取资源文件的数据到内存

2，根据内存的二进制数据，把空壳对象反序列化成实际的对象

3，如果这个对象有依赖其他对象，就递归的去做1和2的操作，直到这个对象完整可用

4，调用对象的初始化函数，并将对象加入到引擎的对象管理中

**资源的卸载**

默认情况下，加载的资源由引擎持有引用，不会被卸载，加载完成后的资源会依赖引擎的gc卸载，如果没有被使用到，会在下次gc的时候释放掉。

但如果不想全局gc导致卡顿，如何处理？

1，大部分UObject可以手动调用ConditionalBeginDestory

2，贴图，材质或Mesh等资源，可以直接调用releaseResource，这里内部会先把贴图或网格内存先释放掉，执行结束之后留下一个空壳UObject

3，RenderTarget，引擎有提供专门的池，可以回收复用。

4，动态材质也有专门的函数删除

5，Actor有DestoryActor，ActorComponent有DestoryComponent

**引用类型**

硬性引用：对象A引用对象B，并导致对象B在对象A加载时加载。

软性引用：对象A通过间接机制引用对象B，可以按照需要加载

构造时引用：在构造时加载资产赋给变量

StaticMesh'/Game/Geometry/Meshes/1MCube.1MCube'

红色：资源类型，加载资源时候可以不写

绿色：资源分区，大部分资源以/Game开头，表示资源在Content目录下，也可以是/Engine

蓝色：相对目录

黄色： 资源的包(Package)名，也就是这个资源所在的真实物理资源文件（uasset/umap）的名字，包其实就是UE4将对象按照自己的规则序列化到磁盘上的文件，在Content Browser里看到的每一个文件都是一个包

紫色： 资源的对象名，因为物理的资源文件里面可能有多个对象，这个名字可以唯一标识包的内部每个对象的唯一名字。比如蓝图资源里有多个UObject，一个关卡文件里有多个Actor，一个UI蓝图里有多个控件。如果不写，UE4的某些接口会默认以包名补充到后面，也就是说默认使用和包名相同的对象名，但有的接口又可能不做处理，所以还是建议写。如果用的不是默认对象，而是资源对象的类，就要在后面加一个\_C，如果是CDO对象，就要在前面加Default\_\_

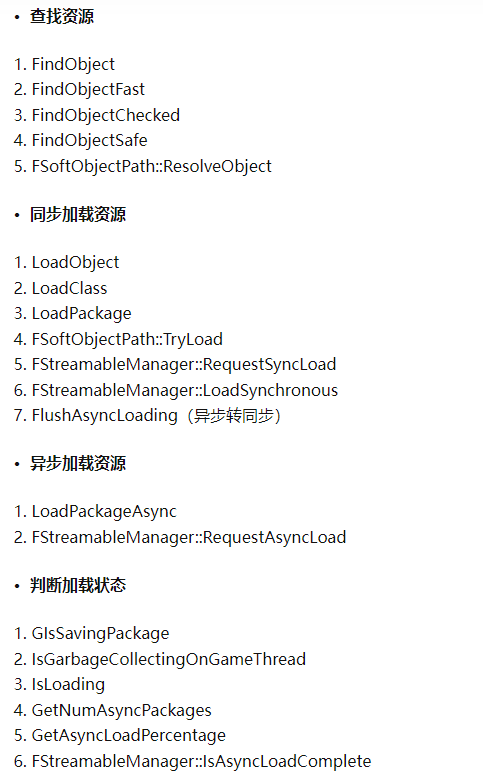
**UAsset的本质**

首先，uasset在磁盘中的表现形式是文件，当我们在引擎里对某个uasset资产做引用的时候，会通过路径将它加载到内存当中，而每个uasset会和UPackage 类对象一一对应，然后存储一个UPackage对象时，会将该包下的所有对象都存到uasset中。而他们之间的关联方式则是我们经常在C++中见到的“Outer”对象，下面是在编辑器模式下打印的一个空Actor蓝图资产的UPackage中关联的其他UObject，包括对应的Class，CDO，蓝图节点等：

IMG_256

这里需要注意的是在编辑器模式下而非Runtime时，因为在不同模式下对应的“Outer”是不同的，编辑器中蓝图UObject的“Outer”是UPackage，而在Runtime是Actor的“Outer”一般是所在的Level。

**业务逻辑要如何加载资源？**



**FindObject，FindObjectFast，FindObjectChecked，FindObjectSafe**

**FSoftObjectPath::ResolveObject**

**这些函数不同版本，最终都会调用到StaticFIndObjectFastInternal这个函数**

在函数内部，ObjectName会被GetObjectOuterHash转为hash值，然后通过Hash值在ThreadHash上取到一个对象列表，UE4会把每个对象存到全局的HashOuter里，这是一个TMultiMap（允许键重复，不能用UPROPERTY标记），也就是一个hash会对应多个value0

**LoadObject，LoadClass，LoadPackage**

会先FindObject在内存中找，找不到再同步加载，会发现不管哪个同步加载函数，最终都会把路径转化为Package再LoadPackage，一个包里如果有多个资源，他们在硬盘上对应的是同一个文件，那么只需加载这个文件就好了。再深入底层可以看到，最终调用的是LoadPackageAsync函数，这是异步加载的入口，最后FlushAsyncLoading，内部阻塞等待，将异步加载转为同步。

这一块还需要好好梳理一下 。。。。没弄特别明白

LoadObject<>的加载（可用于蓝图或者非蓝图）

如，LoadObject<UTexture2D>("ObjectURL")，LoadObject<USoundWave>("ObjectURL\_C")等

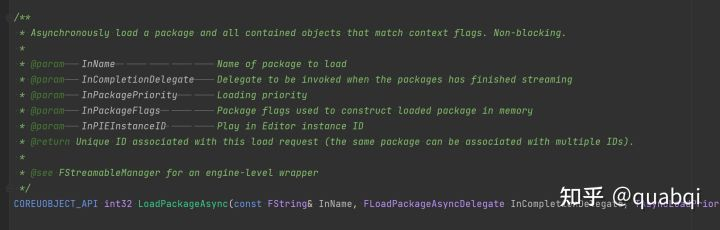
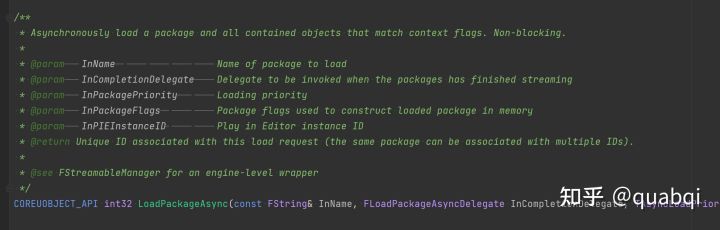
如果需要进行蓝图类对象的加载

LoadObject<UClass>("ObjectURL\_C")

LoadClass用于做类的加载只能用于蓝图

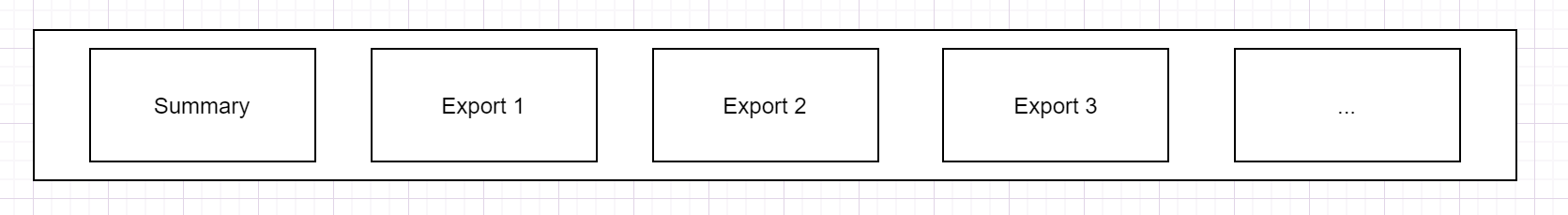
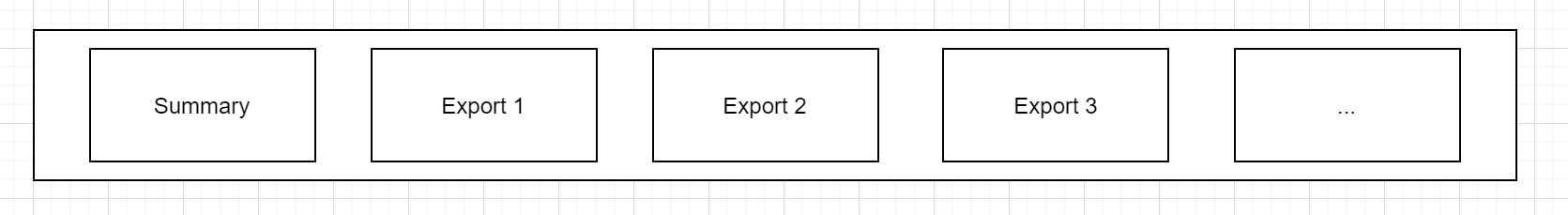
LoadClass<类名>("ObjectURL\_C")

**UE4的资源加载内部是怎样做的？**

****

路径，加载完成的委托等。以及返回一个加载的handle

资源包文件的结构



简单来看，一个UPackage文件就是这样的，有很多UObject序列化的二进制数据

Summary:资源包的摘要信息，是加载资源时最早被加载进来的部分，里面最重要的就是保存有Import表和Export表，Import表描述了这个资源依赖哪些资源，Export表描述了这个包里面有哪些资源，这些资源可以被外面其他资源依赖这样的信息。

Export1，Export2: 这个就是资源包内具体对象序列化后的二进制数据，依次排列着，前面导出表里有多少个资源，这里就会有多少个资源。等所有的Export都加载完成，这个包就加载完成了。

**资源加载的主类**

FAsyncLoadingThread

这个类是负责资源加载的主类，加载逻辑主要是在TickAsyncLoading中执行的

Async：会启动一个专门的加载线程负责Tick资源加载

EDL：在主线程Tick加载。

**资源加载的流程**

1，加载Summary，这一步有IO

2，根据Import表信息，发起依赖资源的加载，并等待所有依赖资源完成

3，加载所有的Export对象，这一步有IO

4，执行PostLoad，并把对象加入引擎管理，完成加载

**EDL(Event Driven Loader)流程**

在EDL流程中把加载流程拆成多个Event，每个Event会做对应的工作，大部分Event结束时会通过入队操作直接开始下个Event

加载资源流程步骤：

Summary阶段： 保存了包内部信息，以及导入表和导出表。在EDL加载的时候，通过CreateLinker启动，会触发io异步读文件，完成时会回调到FinishLinker这一步。在这一步完成后，根据导入表和导出表就知道了当前包自己内部有哪些对象，需要依赖哪些外部对象。这部分在图中每个资源黄色部分只会有1个Node

Import阶段： 等待不包含在包内，但自己包内强引用了一些外部对象，需要等待这些外部对象加载好，才可以加载和反序列化自己的这些对象，继续执行后续步骤。这部分其实就是等待的过程，不触发IO操作。在EDL加载过程中，开始时有多少个Import就会把计数设为多少，之后会把自己的回调函数挂到其他资源上，其他资源加载好了会回调回来把自己的计数减1，计数为0的时候就完成了整个Import步骤，每个资源红框这部分会有多个Node（具体有多少个Import就有多少个Node）

Export阶段： 资源内除了包以外，其他的对象。在引擎EDL加载的时候，因为这些对象是包含在包内的，所以这一步会触发IO，加载等待过程和Import一样，只是多了异步IO，完成时会到PostLoad

PostLoad阶段： 这一步就是调用UObject上的PostLoad函数，程序可以在这一步做一些操作，比如修复资源问题之类

**软引用和硬引用的区别**

在Import阶段时，如果加载的资源依赖了很多资源，需要等待这些资源先加载完成，才能继续加载。这样对性能很不友好。如果不通过依赖关系加载，而是通过业务逻辑自己去按需加载，可以显著提高加载速度。

硬引用：用UProperty标记的那些UObject指针变量，当在反序列化对象时，因为这些变量引用的资源必须跟着一起序列化好才能保证当前的对象是好的，所以这些变量在存储时会反映到Import表里面，写的UProperty资源越多，这个资源加载越慢

软引用：FSoftObjectPath或TSoftObjectPtr引用的资源变量，这些不会随着当前对象一起加载， 但是需要业务逻辑在需要用的时候手动调用加载代码来加载。

因此一个资源Import量很少时，资源加载就会很快。

**资源的卸载**

默认情况下，加载中的资源由引擎持有引用，不会被卸载，加成完成后的资源会依赖引擎的gc卸载。如果没有被使用到，会在下次gc的时候释放掉。

**AssetRegistry**

AssetRegisty可以搜寻uasset文件，并用FAssetData抽象表示，并在有需要时根据FAssetData中的路径线索加载UObject

IMG_263

API坑：GetAssetByObjectPath(const FName ObjectPath, bool bIncludeOnlyOnDiskAssets) ObjectPath不能传应用，传路径，即去掉前面的资源类型